# Path Sequence-Based XML Query Processing

**Ioana Manolescu** [*] — **Andrei Arion**[*] — **Angela Bonifati** [**] — **Andrea Pugliese** [***]

[*] *INRIA Futurs PCRI, FRANCE - firstname.lastname@inria.fr*
[**] *ICAR CNR, ITALY - bonifati@icar.cnr.it*
[***] *DEIS, University of Calabria, ITALY - apugliese@deis.unical.it*

ABSTRACT. *We present the path sequence storage model, a new logical model for storing XML documents. This model partitions XML data and content according to the document paths; and uses ordered sequences as logical and physical structures. Its main advantages are: fast and selective data access plans, and intelligent combination of such plans, based on a summary of the document paths. We validate these benefits through extensive experiments.*

RÉSUMÉ. *Nous présentons le modèle de stockage XML basé sur les chemins du document. Ce modèle sépare la structure du contenu du document, et utilise des séquences comme structures logiques et physiques de stockage. Il permet un accès sélectif aux données, et la combinaison efficace de plans d'accès, en s'appuyant sur un résumé structurel des chemins. Nous démontrons l'intérêt de ce modèle par une série d'experiences.*

KEYWORDS: *XML query optimization, storage model, structural joins*

MOTS-CLÉS : *Optimisation de requêtes XML, modèle de stockage, jointures structurelles*

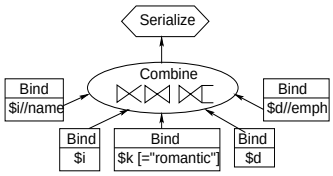



**Figure 1.** *General XQuery evaluation strategy.*

## 1. Introduction

XML data management has been a very active research field lately. The XQuery [xml03] W3C standard for XML querying has almost reached its final state; accordingly, XML database research is concentrating on storage and processing techniques for XQuery evaluation [ALK 02, Y.W 03, TEU 03, BRU 02, HAL 03, CHE 03, MAR 03, NGO 03]. To illustrate the issues involved in XQuery processing, we use the XMark [XMa02] document snippet depicted in Figure 2(a). The document models an on-line auction site: clients, items for sale, and a classification of items in categories.

**Example 1.** Consider the query:

```
for     $i in //asia//item, $d in $i/description where $i//keyword="romantic"
return  <gift> <name> {$i/name} </name> {$d//emph} </gift>
```

For every asian item and every description of this item, such that the item has at least one keyword descendent with value "romantic", the query returns a new gift element with the item name, and the emphasized points in its description. Items with multiple "romantic" keywords must produce only one gift element, with the name and emph descendants if any – or an empty gift element otherwise.

A central notion in XQuery evaluation is that of *variable bindings*. The bindings of a query variable $v$ are the elements (duplicate-free in the sense of element identity, and in document order) found by following the path expressions defining $v$. For example, the bindings for $i consist of the unique, ordered elements found by matching the path expression //asia//item. Similarly, bindings for $d are obtained by following //description from each binding of $i; $i bindings must also be unique, and ordered.

A generic query evaluation strategy in the spirit of [CHE 03] follows from XQuery semantics [xq-]. *Bind* all the variables, and the subqueries in the return clause. *Combine* these bindings through joins. Finally, *serialize* the XML results, as dictated by the return clause. While the previous phases manipulate mainly element IDs, this step requires retrieving and tagging XML element contents.

Figure 1 exemplifies this for the query in Example 1. We purposedly left unspecified the details of each step; we discuss them later. Notice that the joins in Combine pair bindings according to value-based predicates (none in this example), or to *structural relationships* among the element IDs they contain. The latter joins are known as

*structural joins* [ALK 02, CHI 02, BRU 02, ZHA 01]. In Figure 1, within Combine, a structural *join* connects $i and $d bindings; a structural semijoin connects $i and $k, since items with several matching keywords must produce only one result; and structural outerjoins connect $name and emph elements to $i and $d bindings, since a result must be output even for items without name and/or emph descendents.

The performance of XQuery evaluation depends on the efficiency of these three steps: binding, combination, and serialization. The latter can be done efficiently, once the content to be output has been gathered and ordered [SHA 00]. Thus, performance is determined by: bindings variables, according to the query predicates, and the relationships between variables; and combining them.

Recent works provide efficient techniques for implementing [ALK 02] and ordering [Y.W 03, CHE 03] structural joins, relying on a relational or tree-structured storage, augmented with B+-tree indexes [HAL 03, CHE 03]. These indexes provide ordered access to the identifiers of elements of a given tag. We call this storage and indexing approach *tag partitioning* (**TP**).

In this paper, we propose the new *path sequence* storage model. Our model partitions XML content and structure according to the *data paths*, and stores it in *ordered sequences*. Its main advantage is its support for efficient variable binding, up to several orders of magnitude faster than TP. This is due to the precise structural knowledge of the document, included in the path sequence model. Our contributions are:

– We describe the logical and physical *path sequence* storage model. This model is more compact than TP, and allows for efficient document loading. The model comprises a document *path summary*, encapsulating compact structural information.

– We show that based on the path summary, binding variables is much more efficient than when TP is used.

– We extend the iterator execution model, for operators whose output is in document order. Using path summary information, and based on this extension, we derive a family of structural joins algorithms, which use selective predicates on one input to access only the matching part of the other input.

– We show that our storage and query processing model integrates well with XQuery optimization techniques developed for TP [Y.W 03, CHE 03]. Besides more efficient evaluation options, our model also provides precise structural informations to the optimizer, further improving XQuery performance.

The paper is organized as follows. Section 2 describes the path sequence storage model. We then show how to efficiently answer XQuery queries based on this model. Section 3 discusses the construction of variable binding plans. Section 4 addresses binding combination, based on the path summary. Section 5 presents our experimental validation. We discuss related work in Section 6, and conclude in Section 7.

## 2. Path sequence-based storage

We now describe the principles and the implementation of path sequence storage.

### 2.1. *The logical path-sequence storage model*

Our model separates the document structure from the document content, and stores each set of similar items in document order. We decompose the document into three distinct structures, that we describe in turn.

The first structure contains a compact representation of the XML tree structure. We assign unique, persistent identifiers to each element in an XML document. We adopt the [pre, post] scheme used in [ALK 02, CHI 02, HAL 03]. The pre number corresponds to the positional number of the element's begin tag, and the post number corresponds to the number of its end tag in the document. Using this scheme, an element $e_1$ is an ancestor of an element $e_2$ *iff* $n_1.pre < n_2.pre$ and $n_1.post > n_2.post$. For example, Figure 2(a) depicts the [pre, post] ID of each element just above it.

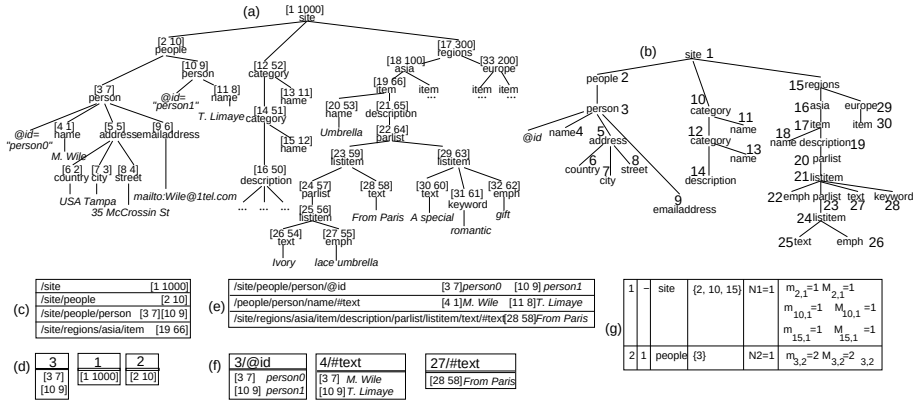



**Figure 2.** *XMark document snippet, its path summary, and some of the resulting storage structures.*

Furthermore, we *partition the identifiers according to the data path* to which the elements belong. Logically, each partition is a *sequence* of identifiers, ordered by their pre field, which reflects the document order. For example, Figure 2(c) depicts a few path sequences resulting from the sample document in Figure 2(a), for the paths /site, /site/people, /site/regions/asia/item etc.

All IDs in a path sequence appear at the same depth in the document tree, which is the path length. Thus, element depth is concisely stored within the paths.

Our second structure stores the contents of XML elements, and values of the attributes. We pair such values to their closest enclosing element identifier (precisely to the pre field of that element ID), and store them in a sequence of [pre, value] pairs ordered by pre. We call such a sequence a *container*.[1] Values found in a container are classified as string, integer, or double values; this can be inferred from their format. For example, Figure 2(e) shows some containers for the document in Figure 2(a), for /site/people/person/@id, /site/people/person/name/#text etc.

The above two structures alone can represent a document without any loss. We added a third indexing structure that will prove very useful in query processing. The *path summary* of an XML document is a tree, whose internal nodes correspond to XML elements, and whose leaves correspond to values (text or attributes). For every simple path $/l_1/l_2/.../l_k$ matching one or several (element or value) nodes in the XML document, there is exactly one node reached by the same path in the path summary.

To each node $x$ in the path summary, we assign an *unique integer path number* which characterizes both the node and the path from the summary root to the node $x$. Thus, each path-driven ID sequence is uniquely associated to a path number. Each container is associated to a pair of a path number, and: either @attrName for attributes, or #text for text content. Figure 2(b) represents the path summary for the XML fragment at its left. Path numbers appear in large fonts next to the summary nodes.

A path summary is different from a dataguide [GOL 97]. The former is always a tree, while the latter can be a graph. Another difference is that a dataguide groups nodes with the same tag, for example, a single dataguide node would stand for the nodes 17 and 30 in Figure 2. We keep separate information about these element sets.

The path summary encapsulates a set of simple and concise *statistics*. Let $x$ be a node in the summary, on a path ending with the tag $t$, and $y$ be a child of $x$. We record:

– $N_x$: the number of elements found on the path $x$ (the size of the ID sequence corresponding to $x$).

– $m_{x,y}, M_{x,y}$: the minimum, resp. maximum number of $y$ children of the XML elements on the path $x$.

*Notations.* For a given document, we denote by $N$ its size, $h$ its height, and $N_{PS}$ the number of nodes in its path summary. We show in Section 5.2.2 that the path summary is very small compared to the document, and thus we keep it in memory at query processing time.

ID sequences, containers, and the path summary together are all the *storage* we materialize from a document, not *indexes* to be added to another persistent storage.

We load an XML document in a single pass, using an event-based parser [xml] which raises events while traversing the document.The algorithm runs in time linear with $N$, using $O(h+N_{PS})$ memory for the stack and path summary.

1. The term is inspired from the XMill project [LIE 00].

### 2.2. *Physical storage for the path sequence model*

An essential feature of the path sequence logical model is order in ID sequences, and in containers. Thus, the physical storage structures implementing it must inherently support order. We consider two ordered persistent structures:

**B+-trees.** This is the option is considered in many works [BRU 02, CHI 02, HAL 03, ZHA 01]. Its advantage is robustness, and good support for updates. Its disadvantage (Section 5.2.2) is the important bloating factor due to extensive ID indexing.

**Persistent sequences.** The alternative that we investigate is the usage of *persistent sequences* as basic storage unit. The advantage of this model is its extreme compactness, which leads to reduced memory usage. We found at least one persistent storage system, endowed with locking and transaction functionality, supporting sequences as first-class citizens [Ber]. A more recent system [LER 03] features a sequence-based storage, largely outperforming an RDBMS in applications where data order is critical (e.g., financial series data). Its drawback is its poor behavior in the presence of updates. In this work, we adopt persistent sequences as physical storage.

Figure 2(d) represents the physical storage of the ID sequences in Figure 2(b). Items in these sequences have constant length, equal to the ID size. Figure 2(f) depicts the sequences resulting from the containers in Figure 2(c). Container entries have variable length, due to the data values. Thus, we use variable-length sequences.

We store the path summary as a sequence of variable-length items, as in Figure 2(g). There is one item per node, comprising: the node number, its parent number, the children numbers, and the statistics previously described. The space occupied by the path summary is linear in $N_{PS}$.

## 3. Binding plans on a path sequence storage

We show how to construct variable binding plans using the path sequence model. Binding plans output *identifiers* of the elements to which the variables are bound.

*Notations.* A *linear path expression* (***lpe***) is an expression of the form: $(v_i?)\ (/|//)\ l_1\ (/|//)\ l_2\ \ldots (/|//) l_k$, where each $l_i$ is a tag or a $*$, the $l_i$s are connected by $/$ or $//$, $k >= 0$, and $v_i$ is an optional query variable. We refer to $k$ as the *length* of *lpe* $P_i$. We call an *lpe* of the form $l_1/l_2/\ldots/l_k$ a *simple lpe*. In particular, each node of the path summary corresponds to the simple *lpe* connecting it to the root. We say that two simple *lpes* are related, when the path summary node of one of them is an ancestor of the other's node. The path summary nodes *matching an lpe* $P$ are those obtained by top-down evaluation of the *lpe* against the path summary, considered as a data tree. We also say the simple *lpes* of these nodes *match* $P$.

We consider a generic tuple-based execution model, following the iterator interface. We denote $op[i]$ the $i$-th column in the output of $op$. Each column may be either of a simple type like string, integer etc., or of type structural ID.

### 3.1. *Binding one query variable*

The first problem we consider is binding a single variable to an *lpe*, such as $x in the query fragment "for $x in /site/people/person/name". Path partitioning allows immediate access to the bindings: just scan the corresponding ID sequence, numbered 4 in Figure 2. We use **IDScan(**$x$**)** iterator, returning the IDs from the path sequence associated to $x$; IDScan also fills in the depth field in element identifiers, with the length of the simple *lpe* of $x$. Its output is ordered, and duplicate-free.

Now consider more complex *lpes*, as for example in the fragment "for $x in //parlist//text". We identify the set of path summary nodes matching //parlist//text. We scan all corresponding ID sequences, and at the same time merge them in a pipelined fashion. The resulting bindings are in document order, and free of duplicates.

In general, we can retrieve the bindings for $x by matching $P$ against the path summary into a set of elementary paths, reading exactly the useful ID sequences, and merging them. No join is required.

Let $x_P$ be the number of nodes in the path summary matching $P$, $N_P$ be the number of bindings for $P$, and $b$ be the blocking factor.

*Using path partitioning, the bindings for $P$ can be obtained with an $I/O$ cost of $O(N_P/b+x_P)$, and a CPU cost of $O(N_P*log(x_P))$. The memory occupancy required is $O(N_{PS}+x_P)$.*

The $I/O$ cost is due to scanning exactly the desired binding IDs. Each ID sequence is physically clustered, but all the $x_P$ matching sequences may not be clustered together, thus the $x_P$ extra reads. The CPU cost corresponds to the merge of $x_P$ ordered ID sequences. The **Merge** operator uses a balanced search tree with $x_P$ leaf nodes. Each ID read from one input is inserted in the search structure in $O(log(x_P))$; to produce an output, the Merge extracts from the search structure the smallest ID it contains, again in $O(log(x_P))$. The memory occupancy corresponds to holding the path summary, and the Merge's search structure.

To match an *lpe* $P$ against the path summary, we traverse the summary top-down, identifying the tags in $P$ from left to right, and adding nodes found to match the last tag in $P$ to the result set. This can be done in $O(N_{PS})$ time. For example, to bind //asia//item, we start from node 1 with $P$, and attempt to match its first tag asia. Since the root has a different tag, we propagate the search for $P$ to all children of the root, which propagate it further. On the regions/asia branch, the asia tag is matched, and the search for //item is propagated to its children etc.

**Binding with simple predicates.** Consider simple selections on the text value, or attributes of $x, as in: for $x in //people//person[@id="person0"]. In such cases, we need to access the container for person/@id. We use a **ContScan(**$x$**)** operator, where $x$ is a container path like 3/@id in Figure 2, returning the ordered [pre,value] container tuples. On top of ContScan, a selection can be applied. If an index on the container ex-

ists, we use the **IdxAccess** operator to access only the desired tuples. We furthermore apply a structural join with IDScan(4), to retrieve [pre,post,depth] IDs.

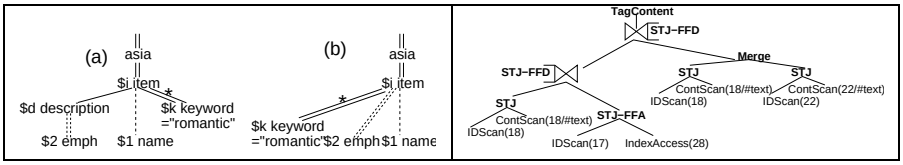



**Figure 3.** *Generalized tree pattern (a) and reduced pattern (b) for the query in Example 1 (left); complete query execution plan to be discussed later (right).*

### 3.2. *Binding several related variables*

**Binding with more complex predicates.** This case can be thought of as binding several variables, on some of which there are simple predicates. We address many-variable binding next. In general, bindings must be computed for several related variables. For example, Figure 3(a) depicts the query variables from Example 1, structured in a *generalized tree pattern (GTP)* [CHE 03]. We use simple lines for $/$ relationships, and double lines for $//$. Dashed lines denote left outerjoin connections between two nodes, considering the parent at left. Edges with a '*' denote left semijoin relationships (also considering the parent at left). We gave the ad-hoc names $1 and $2 to the nodes corresponding to the return clause.

To bind such variables, we proceed in two steps. First, we determine the minimum data sets that need to be accessed to bind the variables, by analyzing relationships among variables, and the path summary. In this step, we may also discover useless variables, or new relationship among them. Second, we construct binding plans, based on the knowledge gathered in the first step.

**Variable path inference.** We start by computing, for every variable, the set of paths in the path summary to which it may possibly be bound. Throughout this section, let $u$, $v$ be query variables such that $u$ is a parent of $v$ in the GTP, and $s_u$, $s_v$ be their respective sets of possible paths.

We compute $s_u$ sets by traversing the GTP and the path summary in parallel, using $s_u$ as a starting point to determine the $s_v$. For example, for the query in Figure 3 and the path summary in Figure 2, $s_i$ is {/site/regions/asia/item}. From the node in $s_i$, we match the /description and $s_d$ is {/site/regions/asia/item/description}. Similarly, matching //keyword from $s_i$ yields the set $s_k$={28} (denoting paths by their numbers in Figure 2(b)). Using $s_d$ and $s_i$, we find $s_1$={18} for $1 and $s_2$={22, 26} for $2.

**Semijoin transformation.** During path inference, for every edge connecting a parent variable $v$ to a child variable $u$, and pair of related paths $x \in s_u$, $y \in s_v$, we compute the *maximum path factor* $Mpf_{x,y}$. This factor is the maximum $M_{x_i,y_i}$ found

by traversing the path summary from node $x$ to node $y$. For example, for the edge between $i and $2 in Figure 3(a), we compute the $Mpf_{17,22}$ and $Mpf_{17,26}$; $Mpf_{17,22}$ is the largest among $M_{17,19}$, $M_{19,20}$, $M_{20,21}$, and $M_{21,22}$.

If $Mpf_{x,y} \leq 1$, and if the edge connecting $u$ to $v$ is a semijoin, we transform it into a join edge, since join and semijoin coincide. For example, if $Mpf_{16,28} \leq 1$ in Figure 2, the execution of the query in Example 1 could start with an access to $k, followed immediately by a join with the bindings for $i. Without this optimization, a duplicate-elimination step on $i would be needed, when combining the bindings.

**Join branch elimination.** Similarly, we compute the *minimum path factor* $mpf_{x,y}$ for each related pair of paths $x$ and $y$, as the minimum $m_{x_i,y_i}$ along the path from $x$ to $y$. If the edge between $u$ and $v$ is a join, $mpf_{x,y} = 1$ and $Mpf_{x,y} = 1$ for all related paths $x, y$, and there is no predicate directly on $v$, then we eliminate $v$ from the GTP, and "glue" all descendents of $v$ to $u$ directly. For example, on the GTP in Figure 3(a), if all asian items have exactly one description, the variable $d is useless and is eliminated, leading to the reduced GTP in Figure 3(b).

**Variable path pruning.** Next, we use each set $s_v$ to prune useless paths from $s_u$, where $u$ is a variable parent of $v$ in the GTP. For example, consider the query: for $p in //parlist, $k in $p//text return $k.

Path inference in this case yields $s_p$={20, 23}, and $s_k$={26}. Since we know by the path summary that 26 is a descendent of 20, but not of 23, we eliminate the useless path 23 from $s_p$, which becomes {20}. Such pruning applies only when in the GTP, $x$ and $y$ are connected by continuous edges only. It does not apply to children connected to their parents by dashed edges, since these children are not required by the query. Thus, even ancestors that do not have them must produce results.

The same pruning may find that a query has no answer, due to an empty path set for a variable connected through joins and semijoins only to the root.

**GTP enrichment.** Finally, new structural relationships among variables may be discovered. For example, consider the fragment "for $i in //item, $p in $i//parlist, $t in $i//text". After path inference and pruning, all paths of $t are descendents of some path of $p. This translates into a new join edge from $p to $t, providing new possibilities of binding combination (the GTP node set does not change).

**The path inference algorithm.** The algorithm is sketched in Figure 4. It consists of a recursive traversal function traverse that is invoked on a path summary node $pv$, given: its parent summary node $pu$; the last GTP node $u$ that has been matched to an ancestor of $pv$; the last GTP *variable* node $x$ that has been mached to an ancestor of $pv$[2]; and the path $px$ where this $x$ was matched.

The invocation of traverse on $pv$ aims to match $pv$ to one of the (GTP) children of $u$. For each such child $v_i$, we check if the tag of $pv$ matches that of $v_i$. If, furthermore,

2. Not all GTP nodes are variables; for example, the asia node in Figure 3 is not a variable, yet it has to be matched before its descendents.

$v_i$ is a variable, then we mark the path we found for it, given that its parent variable $x$ was matched to the path $px$ (line 5). Furthermore, we update $mpf_{px,pv}$ and $Mpf_{px,pv}$ with the $m$ and $M$ statistics stored for $pu$ and $pv$ (line 6). Next, we propagate the matching further by trying to match the children of $v_i$ (in the GTP) to the children of $pv$ (in the path summary), through recursive calls (lines 8-9). For this call, the last variable matched is $x$ (that may have changed when matching $v_i$), and the last GTP node matched is $v_i$. Finally, if the edge beween $u$ and $v_i$ is ancestor-descendent, we also traverse $pv$'s children with the same parameters as for $pv$. This ensures that // GTP edges are correctly matched at any depth in the path summary.

| | Algorithm traverse(GTPNodes $u$,$x$, PSNodes $pu$, $px$, $pv$) |
|---|---|
| 1 | foreach GTP node $v_i$ child of $u$ |
| 2 | $nextX$=$x$; $nextPX$=$px$ |
| 3 | if (the tag of $pv$ matches the tag of $v_i$) |
| 4 | if ($v_i$ is a variable node) |
| 5 | mark($x$, $v_i$, $px$, $pv$) |
| 6 | upd($mpf_{px,pv}$, $Mpf_{px,pv}$, $m_{pu,pv}$, $M_{pu,pv}$) |
| 7 | $nextX$=$v_i$; $nextPX$=$pv$ |
| 8 | foreach PSNode $pw$ child of $pv$ |
| 9 | traverse($v_i$, $nextX$, $pv$, $nextPX$, $pw$) |
| 10 | if (the edge $u$-$v_i$ is ancestor-descendent) |
| 11 | foreach PSNode $pw$ child of $pv$ |
| 12 | traverse($u$, $x$, $pv$, $px$, $pw$) |

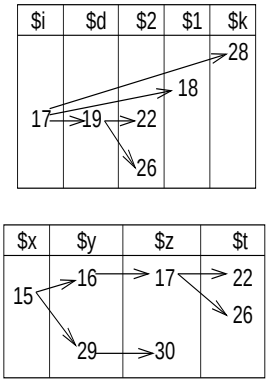



**Figure 4.** *The path inference algorithm, and sample results.*

The inference process outputs the path lists found for each variable, grouped hierarchically following the variables in the GTP. In Figure 4, at the top right, we depicted the result for the query in Example 1. Each path found for a given variable points to the corresponding paths found for its children variables in the GTP. (Arrows preserve all edge semantics from the GTP; arrows leaving from the same path of a variable, and reaching different paths of another variable, have *or* semantics.) Arrows are also annotated with the $mpf$ and $Mpf$ computed factors.

Paths without required descendents are pruned at this point. For example, in Figure 4 at the bottom right, we show the paths resulting for the variables in: "$x in //regions, $y in $x/*, $z in $y/item, $t in $z//emph" on the path summary in Figure 2. Realizing that path 30 for $z has no descendent path for $t, we prune 30, and then 29.

Semi-join transformation and join branch elimination also apply (and may reduce) on the result of path pruning. Finally, to infer new edges in the GTP, we analyze the surviving variable paths. This process may be piggybacked on traverse, using a matrix of $N_{PS}^2$ bits in which we mark which path is descendent of which other.

**Binding plan construction.** Using the path sets that result from the above steps, we construct individual binding plans, as described in Section 3.1. For the query in Example 1, this yields: IDScan(17) for $i; Join(IDScan(28), Filter(ContScan(28/#text)) for $k; IDScan(19) for $1; Merge(IDScan(22), IDScan(26)) for $2 ($d has been eliminated). These plans are selective, as they only access IDs whose paths allow them to contribute to the query result.

**Metadata and statistics for the binding plans.** Let $|op|$ denote the cardinality estimate for operator $op$. If $op[i]$ is of type ID, we denote by $pList(op, i)$ the set of path numbers on which IDs from $op[i]$ may be found. For IDScan($x$) plans, $pList(op)$={$x$}, $|op|=N_x$. For plans of the form Merge(IDScan($x_1$),. . ., IDScan($x_k$)), $pList(op)$={$x_1$,$x_2$,. . . $x_k$}, we estimate $|op|$ as $\sum_{x \in pList(op)} N_x$. For Filter or IdxAccess plans, predicate selectivity estimation are needed to estimate $|op|$. Value statistics on containers can be used to this purpose.

## 4. Combining binding plans

Binding plans are combined bottom-up through joins and structural joins, as in [Y.W 03, CHE 03]. We now focus just on efficient techniques enabled by the path sequence model, which can be profitably integrated to the combination step.

Section 4.1 presents an order-preserving condition for structural joins. Section 4.2 describes more complex join algorithms enabled by our ordered path-partitioned storage. We then briefly discuss query plan construction.

### 4.1. *Order-preserving structural join*

The *StackTreeDesc (STD)* and *StackTreeAnc (STA)* algorithms [ALK 02] directly apply in our context. They pair inputs by structural relationships among structural [pre, depth, post] element IDs they contain. STA and STD require inputs sorted in the document order of the IDs to join; STA preserves the order of the ancestor IDs, and STD the order of the descendent IDs. Both algorithms employ a stack; STA needs some other data structures. Both work in pipeline.

To avoid sorting steps, it would be interesting to have a structural join operator preserving *both* ancestor and descendent ID order; however, this is not always possible. A structural join preserves both input orders iff the IDs in the ancestor are free of ancestor-descendent pairs [ALK 02, CHE 03]. This condition is data-dependent, and thus impractical to check. We provide a sufficient, easier to check condition:

*Let $op_1$ and $op_2$ be two operators, such that $op_1[i]$ and $op_2[j]$ contain element IDs. If the paths in $pList(op_1, i)$ are all pairwise unrelated, then a structural join matching ancestor IDs in $op_1[i]$ with descendent IDs in $op_2[j]$ can preserve both input orders.*

Given a set $pList$ of size $k$, the condition can be checked on the path summary in $O(k * h)$: from each node in $pList$, we navigate upwards to the root; if we encounter

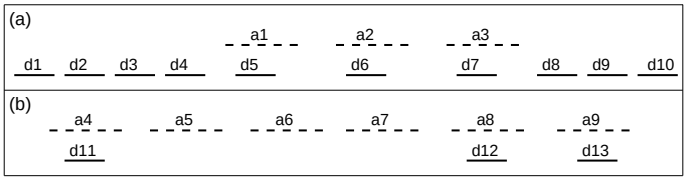



**Figure 5.** *Structural joins when inputs can be skipped.*

another node from $pList$, then the paths are not pairwise unrelated. Thus, when $anc$ paths are unrelated, the (simpler) STD algorithm can be used with the knowledge that it preserves both ancestor and descendent order.

### 4.2. *The STD-FastForward family of algorithms*

A structural join may avoid reading parts of the inputs. Figure 5 depicts two such examples. Element IDs are shown as horizontal [pre,post] segments, dashed for IDs of the $anc$ operator, and solid for IDs of the $desc$ operator. An $anc$ ID $a$ is an ancestor of a $desc$ ID $d$ iff the segment of $d$ is under the segment of $a$.

In Figure 5(a), the first descendent of $a_1$ is $d_5$. Thus, $d_1 - d_4$ could be skipped, since they do not produce join results. Instead, the join should advance $desc$ to the first $d$ that may be a descendent of $a_1$, that is, such that $d$.pre$>a_1$.pre. Since the $desc$ tuples are sorted on pre, it is possible to advance $desc$ according to this order.

In Figure 5(b), after the pair $a_4, d_{11}$, the join could skip $a_5 - a_7$, and advance $anc$ to the first $a$ that could be an ancestor of $d_{12}$, that is, such that $a$.post$>d_{12}$.post. This requires skipping on post, but the inputs are ordered on pre; the skip is only possible if the two orders coincide. From the pre and post semantics, it is easy to see that these orders coincide in an ID set iff it has no ancestor-descendent pairs.

A sufficient condition applies: Given an operator $op$ such that $op[i]$ is of type ID, if $pList(op, i)$ are pairwise unrelated, then the pre and post orders in $op[i]$ coincide.

Thus, we design the *STJ-FastForward* (**STJ-FF**) structural join algorithm; it is derived from StackTreeDesc, although for readability we omitted the stack manipulation details. STJ-FF skips inputs both based on $anc$.post and $desc$.pre. It relies on two new methods provided by $anc$ and $desc$:

– *nextFF-pre(*$col$*, int* $n$*)* positions an operator $op$ to the first output tuple having $op[col]$.pre$\geq n$. If such a tuple does not exist, nextFF-pre returns false.

– *nextFF-post(*$col$*, int* $n$*)* positions an operator $op$ to the first output tuple having $op[col]$.post$\geq n$. If such a tuple does not exist, nextFF-pre returns false.

Using these methods, a parent operator commands its children to produce tuples with certain values only. They can only be called on operators $op$ whose output is

sorted on $op[col]$.pre; nextFF-post further requires that the paths in $pList(op, col)$ be pairwise unrelated.

IDScan and ContainerScan operators implement the nextFF methods by binary search over the sequence. The Merge, and the STJs, implement it combining calls to the nextFF methods of their child operators. An STJ supports: ($i$) nextFF-pre on both the ancestor and descendent ID columns; ($ii$) nextFF-post on the ancestor and descendent ID columns, *if* those paths are unrelated. For example, let $op$=STJ(IDScan(17), Merge(IDScan(21), IDScan(24))) in Figure 2. $op$ supports: nextFF-pre on columns $0$ and $1$, and nextFF-post on $0$ only, since 21 and 24 are related.

In Figure 5(a), after the initial skip to $d_5$, the region $d_5$, $d_6$, $d_7$ contains useful IDs. Thus, after finding $d_5$, a binary search to find $d_6$ would make useless effort. Therefore, we implement skipping "optimistically" in the IDScan and ContScan operators: the next available value is examined, and only if it doesn't satisfy the skip condition, binary search is performed.

Skipping in structural joins may lead to thrashing [ZHA 01]. For skipping to be profitable, it must be used only above a certain useless ID frequency. If a block is read for only one useful ID, skipping the comparisons for the other ones will not bring noticeable benefits (Section 5.3).

The situations in Figure 5 do not always occur simultaneously. For example, consider "for $x in //person[id="person0"], $y in $x/name". When joining bindings for $x and $y, there will be many useless $y bindings, and a single (useful) one for $x. Thus, we specialize STJ-FF into: **STJ-FFA**, which only skips $anc$ tuples, and advances $desc$ through next() calls only; and **STJ-FFD**, which only skips $desc$ tuples, and advances $anc$ through next(). We demonstrate their interest in Section 5.3.

**Optimizing with structural joins and path sequences.** To evaluate a given structural join, the optimizer examines the operands' $pLists$, and whenever ancestor $pLists$ are unrelated, it knows that both ordered will be respected. For example, *all* structural joins derived from the XMark queries feature unrelated $anc$ paths. The optimizer also needs to decide when to use skipping structural joins as those described in Section 4.2. Cardinality estimates on binding plans, and structural join plans, are used. For example, for the query: "for $x in //person[@id="person1"], $y in $x//age", STJ-FFA is a good join candidate, since $x is only bound to a single person.

## 5. Experimental validation

In this section, we assess the performance of the path sequence model: storage conciseness and binding plans (Section 5.2) and binding plan combination (Section 5.3).

### 5.1. *Hardware and software environment*

We implemented the path sequence model in our Java-based Gex prototype; it consists of about 150 classes, or 40.000 lines of code. The persistent storage is built on the BerkeleyDB [Ber] database library. We used a DELL D800 laptop, with a Pentium IVM at 1.4 GHz and 1Gb of RAM, running RedHat Linux 9.0. All times are averaged over 20 hot runs. We report results on: an **XMark** document of 118 Mb; the **DBLP** data set of 133 Mb; and the **SwissProt** dataset of 114 Mb.

### 5.2. *Data storage and binding plans*

#### 5.2.1. *Comparison baseline: tag partitioning*

We compare our approach with the state-of-the-art XML storage model, based on partitioning, and structural identifiers. Within this model, XML documents are stored as persistent trees or relational tables. Redundant B+-tree indexes are added to the storage, indexing all element IDs by a (tag+pre) key. These indexes give access to the list of all IDs of a certain tag. We call this approach *tag partitioning* (**TP**); it is used in [CHE 03, HAL 03, ZHA 01]. We refer to our approach as *path sequences* (**PS**).

In TP systems, index-based query processing is the best execution strategy. Thus, we compare our query processing techniques with the techniques from [ALK 02, Y.W 03, CHE 03] based on the tag indexes. To that purpose, we implemented TP also, in two variants: B+-trees, and by tag-based sequences. The latter can be seen as merging all sequences (or containers) ending in the same tag, in a single sequence.

#### 5.2.2. *Space occupancy*

Figure 6 compares the space occupancy for PS, and TP using B+-trees as in [CHE 03, HAL 03, ZHA 01]. For XMark, document structure stored with PS takes about 10% of the document size; with TP, it takes 20 times more. This is due to the presence of tags in the B+-tree keys (with PS, they are only stored in the path summary), and to the intermediary index pages. Overall, the document size is multiplied by 3.5-4 by when stored with TP, close to the factor of 4 reported in [CHE 03]; with PS, the storage size is slightly smaller than the document. The path summary occupies 26 Kb for XMark, 6 Kb for DBLP, 10 Kb for SwissProt; $N_{PS}$ is respectively 514, 122, and 117. This validates the assumption that it can be kept in memory when processing queries. Having demonstrated sequence conciseness, in the following, we report on PS *and* TP performance on a persistent sequence-based storage.

#### 5.2.3. *Variable binding performance with TP and PS*

Without loss of generality, we consider *lpes* where tags are connected by //.

The case most favorable to TP is a variable bound to a *lpe* of the form //tag: PS merges several IDScans (Section 3.1), while TP only looks up the IDs of the given tag. When only one path summary node matches //tag, PS and TP coincide.

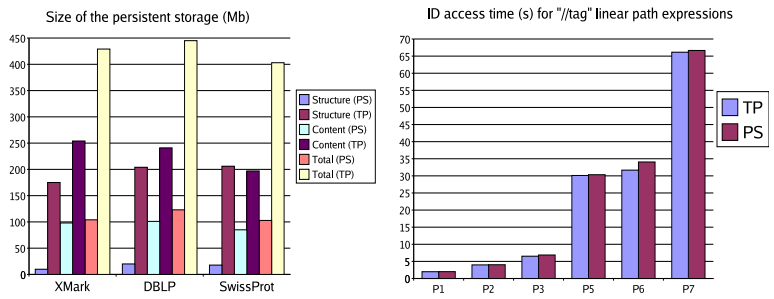



**Figure 6.** *Space occupancy comparison for PS and TP (left), PS merging overhead when binding one variable (right).*

Figure 6 compares the execution time for binding plans when the TP and PS plans differ. The *lpes* used are: $P_1$=//item, $P_2$=//description, $P_3$=//bold, $P_4$=//category on XMark; $P_5$=//title, $P_7$=//author on DBLP; and $P_6$=//Descr on SwissProt. The cardinalities of the binding plans range from 2.000 ($P_1$) to 720.000 ($P_7$). The PS overhead, due to scanning and merging different sequences, is negligible. This is because: ($i$) the Merge is pipelined, CPU-only, and efficient due to the ordered tree of inputs it maintains; ($ii$) IDs are tightly packed by PS in sequences and read sequentially, thus no thrashing occurs; ($ii$) the Merge fan-out is of the order of hundreds, making it possible to keep in memory one block from each sequence to be merged.

**Binding a variable to longer *lpes*.** In such cases, TP is handicapped by its unselective data access. For example, let $P$ be /site/people/person/name: TP only allows to access all name IDs. Most of such IDs are *not* on the required path; they may be on paths 11, 18 etc. (Figure 2), leading to useless data scans. To separate just the person names, we must access all person IDs (thus read even more data !), and apply a structural join between the two. Furthermore, we must read all site and people IDs, and successively join with them also, to separate just the proper names. In general, to bind a variable to a *lpe* of length $k$, TP must scan the IDs of all tags involved, and compute an $k$-way join. PS just scans the result. For the query "for $x in //parlist//text", TP must join all parlist IDs with all text IDs; but a new complication arises. A text ID may be a descendent of two parlist elements, as is the case for the element [26,54] in Figure 2. Such text elements will appear twice in the join result. Thus, a final duplicate elimination (*dup-elim*) is required.

**Optimized TP.** To give TP a chance, we use the PS path summary to eliminate useless data scans, joins, and dup-elims from TP binding plans. We call *optimised TP (OTP)* the most efficient building plan derived this way from a TP stage.

Figure 7 (left) considers *lpe*s of length 2: $P_8$=//description//parlist, $P_9$=//parlist//listitem, $P_{10}$=//item//keyword, $P_{11}$=//person//name, $P_{12}$ = //cate-

gory//name on XMark; $P_{13}$=//article//title, $P_{14}$=//book//author and $P_{15}$=//book//title on DBLP; $P_{16}$=//METAL/Descr and $P_{17}$=//LIPID//Descr on SwissProt. For $P_8$ and $P_9$, PS and OTP coincide; TP performs one extra join and a dup-elim. From $P_{10}$ to $P_{17}$, OTP only spares the dup-elim, which is cheap since the join result order was favorable. The performance difference between PS and OTP reflects the proportion of useless IDs scanned by OTP due to the tag-partitioned storage. This proportion reaches 430 for $P_{14}$; PS is very fast for $P_{14}$, $P_{15}$ and $P_{17}$.

In Figure 7 (right) we measured *lpe*s of length 3. (O)TP performance is determined by the data scans; PS speed-up is of up to several hundreds. The TP handicap grows with the length of the *lpe*.

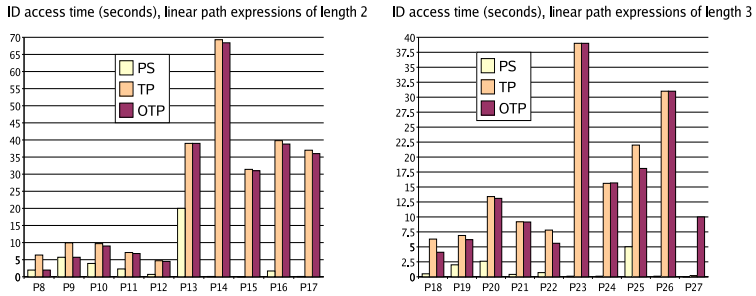



**Figure 7.** *Binding one variable to* lpe*s of length 2 (left), and 3 (right).*

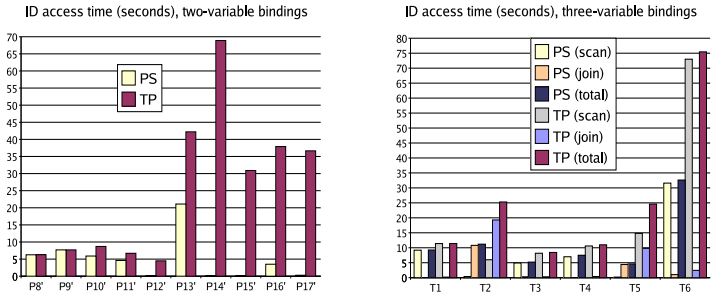
### 5.3. *Combining variable bindings*

Figure 8 compares TP and PS performance when binding two, respectively, three related variables. In these cases, TP cannot benefit from any optimization, since the scans and the structural join are required, in order to get binding pairs. Furthermore, PS must also scan many inputs and join them. Also, in Figure 8, PS and TS use the best possible join order. In Figure 8 (left), we consider again the paths $P_8$ to $P_{17}$ that we used in Figure 7(left), but this time we replace "for $x in //tag$_1$//tag$_2$" with "for $x in //tag$_1$, $y in $x//tag$_2$". For example, in Figure 8(top), $P'_8$ stands for the two-variable pattern "for $x in //description, $y in $x//parlist". We notice that for $P'_8$ and $P'_9$, PS and TP are very close, since: TP makes no useless reads (all parlist elements are under description elements, and similarly for $P'_9$), and the cost is dominated by data access. For $P'_{10}$ - $P'_{17}$, the unselective scans of TP translate directly into huge performance differences with PS; the curves mirror quite closely those from Figure 7 (left). The biggest time we measured for the join in itself was 4.2 seconds, for $P'_{17}$.

In Figure 8 (right), we consider three-variable patterns, $T_1$-$T_5$ on XMark and $T_6$ on DBLP, shown just underneath the corresponding graph. While quite small, these

patterns illustrate well the performance issues involved. We have chosen just 1-tag *lpes* to connect $x to $y and $z, to pick the best possible case for TP: if longer paths connect variables, TP will do useless scans and joins, as previously explained.

Overall, scan, join, and total time are still smaller for PS than for TP. For $T_1$, $T_3$, $T_4$ and $T_6$, the join cost is negligible for both, compared to the data scan cost. In these cases, simply using a PS storage provides for faster binding. For $T_2$ and $T_5$, the joins by themselves take a more important part; even with the best ordering, the first join creates many intermediary results, on which the second join spends more time. This problem can be alleviated using holistic twig joins [BRU 02], which reduce intermediary results in large structural join trees.




| | |
|---|---|
| $T_1$ | $x in //person, $y in $x//name, $z in $x//watch |
| $T_2$ | $x in //asia, $y in $x//item, $z in $x//description |
| $T_3$ | $x in //item, $y in $x//description, $z in $x//parlist |
| $T_4$ | $x in //item, $y in $x//parlist, $z in $x//keyword |
| $T_5$ | $x in //categories, $y in $x//listitem, $z in $x//text |
| $T_6$ | $x in //article, $y in $x//title, $z in $x//year |

**Figure 8.** *Binding 2- and 3-variable patterns.*

We have also performed measures with skipping join algorithms (STJ-FFA or STJ-FFD); we conclude that they are profitable when one inputs size is very small (e.g. point queries). We also tested reconstruction plans and found a good linear scaleup (omitted for space reasons).

**Conclusions** The PS storage model is very compact, since it has practically no overhead; it supports variable binding up to several orders of magnitude faster than TP, due to its precise structural knowledge. The ordered PS model enables skipping useles data, at the level of joins, but also larger plans. The target of PS optimizations (selective data access) and techniques like [BRU 02] are complementary, thus these optimizations can be fruitfully combined to improve query performance.

## 6. Related work

Our work is placed in the context of XQuery processing based on structural joins. The closest related works have been done within the Timber and Niagara systems [ALK 02, Y.W 03, BRU 02, CHE 03, HAL 03], and are based on tag partitioning. A path-partitioned storage provides much better performance for variable binding plans, as explained in Section 5.2.1.

Our join branch elimination (Section 3.2) is an instance of constraint-based minimization [AME 01]; $m_{x,y}$ and $M_{x,y}$ act as child constraints, while $mpf_{x,y}$ play the role of descendent constraints. More recent works studied XPath containment and simplification using schema information, such as a DTD or an XMLSchema [WOO 03]. While our path summary resembles a schema, there are some informations that a schema provides, while a path summary doesn't. Thus, schema-driven techniques could be applied before/during our path inference process. The path summary is easy to construct, concise, and very useful for XQuery processing, especially for variable binding. Interestingly, 40% of the XML documents found on the Web have a DTD [MIG 03], and about 1% have an XML Schema; without schemas, path summaries still apply.

Techniques for input skipping in structural joins on pre were described using B+-tree indexes [ZHA 01, CHI 02], chained stacks [BRU 02], and spatial indexing methods [TEU 03]. Our STJ-FF avoids the same comparisons as in [ZHA 01, CHI 02], without a storage overhead, as we incorporate skipping into the execution model directly. Our first contribution in terms of skipping useless ID comparisons is path inference, which allows to construct joins on IDs from related paths only. This optimization can be composed with those in [TEU 03, BRU 02]. The second contribution is skipping on post, under path-related conditions.

## 7. Conclusions

We have presented the path sequence storage model, a compact and fragmented model for XML storage. The path sequence model improves over similar systems by preserving precise path information, in the storage as well as under the form of a path summary that is extensively used for query optimization. Our model is fully implemented within the GeX system; we validated its performance advantages through a series of experiments.

**Acknowledgements.** The authors are extremely grateful to Alberto Lerner, for many useful comments on this paper.

## 8. References

[ALK 02] AL-KHALIFA S., JAGADISH H., PATEL J., WU Y., KOUDAS N., SRIVASTAVA D., "Structural Joins: A Primitive for Effi cient XML Query Pattern Matching", *ICDE*, 2002.

[AME 01] AMER-YAHIA S., CHO S., ET AL. L. L., "Minimization of Tree Pattern Queries", *SIGMOD*, 2001.

[Ber] "The BerkeleyDB database library", http://www.sleepycat.com.

[BRU 02] BRUNO N., KOUDAS N., SRIVASTAVA D., "Holistic twig joins: Optimal XML pattern matching", *SIGMOD*, 2002.

[CHE 03] CHEN Z., JAGADISH H., LAKSHMANAN L., PAPARIZOS S., "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery", *VLDB*, 2003.

[CHI 02] CHIEN S., VAGENA Z., ZHANG D., TSOTRAS V., "Efficient Structural Joins on Indexed XML Documents", *VLDB*, 2002.

[GOL 97] GOLDMAN R., WIDOM J., "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases", *VLDB*, Athens, Greece, 1997.

[HAL 03] HALVERSON A., BURGER J., GALANIS L., KINI A., KRISHNAMURTHY R., RAO A., TIAN F., VIGLAS S., WANG Y., NAUGHTON J., DEWITT D., "Mixed Mode XML Query Processing", *VLDB*, 2003.

[LER 03] LERNER A., SHASHA D., "AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments", *VLDB*, 2003.

[LIE 00] LIEFKE H., SUCIU D., "XMILL: An Efficient Compressor for XML Data", *SIGMOD*, 2000.

[MAR 03] MARIAN A., SIMEON J., "Projecting XML Documents", *VLDB*, 2003.

[MIG 03] MIGNET L., BARBOSA D., VELTRI P., "The XML Web: A First Study", *Proc. of the Int. WWW Conf.*, 2003.

[NGO 03] NGOC T. T. D., GARDARIN G., "Evaluating XQuery in a full-XML mediation architecture", *Proc. Journées Bases de Données Avancées*, 2003.

[SHA 00] SHANMUGASUNDARAM J., SHEKITA E., BARR R., CAREY M., LINDSAY B., PIRAHESH H., REINWALD B., "Efficiently Publishing Relational Data as XML Documents", *VLDB*, Cairo, Egypt, 2000.

[TEU 03] TEUBNER J., GRUST T., VAN KEULEN M., "Bridging the GAP between Relational and Native XML Storage with Staircase Join", *VLDB*, 2003.

[WOO 03] WOOD P., "Containment for XPath Fragments under DTD constraints", *Proc. of the Int. Conf. on Database Theory (ICDT)*, 2003.

[XMa02] "The XMark benchmark", www.xml-benchmark.org, 2002.

[xml] "The SAX project", www.saxproject.org.

[xml03] "The XQuery Language", www.w3.org/TR/xquery, 2003.

[xq-] "XQuery 1.0 Formal Semantics", www.w3.org/TR/2004/WD-xquery-semantics.

[Y.W 03] Y. WU J. PATEL H. J., "Structural join order selection for XML query optimization", *ICDE*, 2003.

[ZHA 01] ZHANG C., NAUGHTON J., DEWITT D., LUO Q., LOHMAN G., "On Supporting Containment Queries in RDBMSs", *SIGMOD*, 2001.